Exploring Functional Acceleration of OpenCL on FPGAs and GPUs Through Platform-Independent Optimizations

Umar Ibrahim Minhas, Roger Woods, and George Karakonstantis

Queens University Belfast

Abstract. OpenCL has been proposed as a means of accelerating functional computation using FPGA and GPU accelerators. Although it provides ease of programmability and code portability, questions remain about the performance portability and underlying vendor's compiler capabilities to generate efficient implementations without user-defined, platform specific optimizations. In this work, we systematically evaluate this by formalizing a design space exploration strategy using platform-independent micro-architectural and application-specific optimizations only. The optimizations are then applied across Altera FPGA, NVIDIA GPU and ARM Mali GPU platforms for three computing examples, namely matrix-matrix multiplication, binomial-tree option pricing and 3-dimensional finite difference time domain. Our strategy enables a fair comparison across platforms in terms of throughput and energy efficiency by using the same design effort. Our results indicate that FPGA provides better performance portability in terms of achieved percentage of device's peak performance (68%) compared to NVIDIA GPU (20%) and also achieves better energy efficiency (up to 1.4×) for some of the considered cases without requiring in-depth hardware design expertise.

1 Introduction

The rapidly increasing use of heterogeneous accelerators such as Graphic Processing Unit (GPU) and Field Programmable Gate Array (FPGA) in data centres necessitates the adoption of a unified programming environment that also maintains better throughput and energy efficiency [1]. This is hard to achieve, however, due to widely varied architectures and technologies, which have been traditionally programmed via specialized languages e.g. VHDL for FPGAs and CUDA for NVIDIA GPUs, using detailed knowledge of underlying hardware. In addition to programming inefficiency, this hinders fair comparison of achieved performance and design cost across the different accelerating technologies.

To address this challenge, the Open Computing Language (OpenCL) [2] has been introduced as a C-based platform-independent language, to allow parallelism to be expressed explicitly regardless of the underlying hardware. OpenCL is now supported by a range of programmable accelerators including GPUs and FPGAs. However, OpenCL only provides functional portability and the application implementation needs to be optimized by the underlying accelerator vendor compilers. Under such a reality, the question remains about performance portability of OpenCL applications on various accelerators. That is, how much performance an application can achieve across various platforms and how to gauge the efficiency of the vendor-specific compilers to map OpenCL source code to the targeted device with minimum or even no user-defined platform-specific optimizations. Also, it is questionable if FPGAs still require more in-depth knowledge of underlying hardware compared to other technologies.

Achieving performance portability and fair evaluation is becoming extremely important with increased usage of accelerators in data centres and cloud environments [3]. Researchers have approacehd these challenges from two angles. On one hand some studies compare programming languages such as a hardware descriptive language (HDL) and Compute Unified Device Architecture (CUDA) with OpenCL on the same platform, e.g. FPGAs [4] and GPUs [5]. On the other hand there is portability evaluation of the same language e.g. OpenCL across multiple platforms i.e. NVIDIA GPU, AMD GPU, Intel CPU and Sony/Toshiba/IBM Cell Broadband Engine [6]. These works conclude that although platform independent language can lead to better portability, additional effort is required for tuning kernels to each device to achieve comparable performance.

An architectural and programming model study on fractal video compression involving optimization of OpenCL on FPGA has been presented in [7] and provides a series of FPGA-based optimizations on FPGA before comparing the results with CPU and GPU for an optimized kernel. In [8], six benchmarks of the Rodinia suite are evaluated using OpenCL and FPGA-specific optimizations are performed on kernels optimized for GPU-like devices, achieving up to 3.4x better energy efficiency compared to GPUs. However, this work requires platform-specific optimizations, which partially nullifies the motivation behind a software-based approach via a unified programming environment. In addition, they compare the output with already-optimized implementations on other platforms and do not discussed performance portability.

In this paper, we develop and apply a systematic approach to gauging performance portability and fair evaluation. We apply a set of uniform micro-architectural optimizations for fair porting, optimization and evaluation of applications across platforms using OpenCL. The optimizations are based on carefully selected common micro-architectural features that can be easily parametrized via the OpenCL model. Initially, we take C source code of kernels for 3 accelerated computing applications, namely matrix-matrix multiplication (SGEMM), Binomial-tree Option Pricing (BOP) and 3 dimensional Finite Difference Time Domain (FDTD) and port them to OpenCL as base kernels before applying the platform-independent optimizations.

We then evaluate these optimisations on 3, state-of-the art, platforms namely Altera FPGA, a high performance NVIDIA GPU and a low power ARM Mali GPU. In doing so, we analyse the underlying compilers' job in generating an optimized implementation. We also compare the achieved performance to the theoretical peak throughput and platform-specific implementations. To the best of our knowledge, this is the first work to discuss platform-independent, design space exploration across FPGA and GPU and use the same optimization efforts, programming environment and runtime for a fair comparison of heterogeneous accelerators for throughput and energy efficiency.

In brief the key contributions of this paper include:

– A micro-architectural optimization and design space exploration approach based on platform-independent parameters of OpenCL model.
– Implementation of three applications from linear algebra (SGEMM), financial computation (BOP) and electromagnetic modelling (FDTD) on heterogeneous accelerators while analysing the architectural and algorithmic challenges using OpenCL.
– Fair comparison of the implementations of the above applications on state-of-art FPGA and GPUs in terms of application-specific metrics for throughput and energy efficiency while trying to keep the design efforts the same.
– Comparison of achieved performance through platform-independent optimizations with theoretical peak and platform-specific optimizations.

The rest of the paper is organized as follows. Section II describes the design environment by reviewing OpenCL programming model, the tested use cases and experimental platforms. Section III summarizes the optimization and design space exploration methodology and its application to the use cases. Section IV and V then analyse the throughput and energy efficiency optimizations on each of the platforms. Finally, Section VI concludes the presented work.

2 Design Environment

2.1 Overview of OpenCL

OpenCL is a cross platform open standard for heterogeneous parallel programming that defines platform-independent APIs for abstraction of parallelism. A serial OpenCL program runs on a host CPU with parallel compute intensive tasks being offloaded via a kernel definition and OpenCL runtime onto a compute device. A compute device contains one or more compute units (CU) each of which has one or more processing elements (PE) (Fig. 1).

A complete application is usually divided into smaller tasks, work-groups, with each running independently on a CU. A work-group has further 3 dimensional parallel work-items, which run on a PE. Memory types of OpenCL based on latency are high-latency *global* memory accessible to a whole kernel and fast *local* memory shared within a work-group.

2.2 Use Cases

We take three different use cases from linear algebra, financial computation and electromagnetic modelling; their computation range offers diverse testing of micro-architecture, as summarized in Table 1. These are briefly explained below.

Matrix-Matrix Multiply: SGEMM of two square matrices, A and B, of order n results in a matrix C of order n where each ij^{th} element of C is the dot

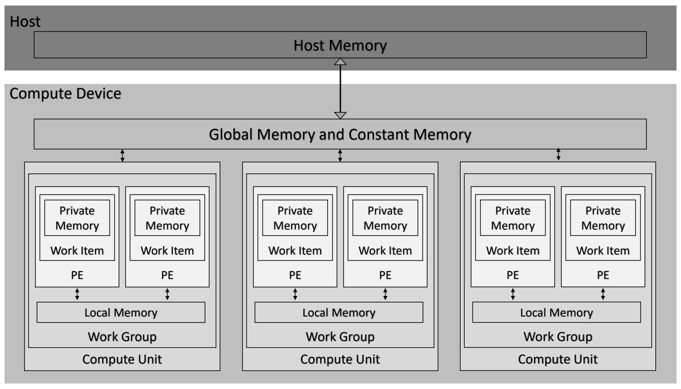

Fig. 1. OpenCL architecture

Table 1. Use cases characteristics

Characteristic	SGEMM	BOP	FDTD
Domain	Linear Algebra	Finance	Modelling
Evaluated Data Points	Up to 32K	Up to 4K	105M
Data Dimensions	1	2	3
Data Access Pattern	Regular	1 Regular, 1 Irregular	Regular
Architecture	SIMD	Iterative	Sliding Window
Data Reuse	Order of matrix	2	$Radius^2$

product of i^{th} row of A and j^{th} column of B. SGEMM is a main component of various libraries and benchmarks (such as LAPACK) used in dense linear algebra algorithms and for benchmarking purposes.

Binomial-Tree Option Pricing: BOP is a key model in finance that offers a generalized method for option valuation and can be applied for more exotic options with complex features. It calculates the value of the option at the final nodes of a binomial tree. The next, computationally complex step involves walking backwards up the tree calculating the price of all nodes at each time step sequentially, until the first node is reached. Each node, n, in a vertical column at a time step, t, is dependent on two nodes, n and $n+1$, in the time step, $t+1$.

3 Dimensional Finite Difference Time Domain: FDTD is an important numerical method in electromagnetic numerical modelling which builds a model space and stores it in memory. The calculation of electric and magnetic field progression in 3D space, $dimx \times dimy \times dimz$, is conducted in a sliding window fashion, where window is a sphere of set radius. Apart from the computational needs, another reason for selecting this algorithm for comparison is due to the availability of fine tuned implementations from vendors, Altera and NVIDIA.

2.3 Platforms

We evaluate 3 typical state-of-the art platforms, based on the same technology, from Altera, NVIDIA and ARM (Table 2). In terms of architecture, both GPUs share similarities with fixed micro-architecture consisting of processing cores and cache. The main differentiation factor lies in the device scale and non-availability of separate local memory for CUs in Mali. In comparison, FPGA offers a reconfigurable architecture with variable precision Digital Signal Processors (DSPs), a common large local memory for all DSPs and non-availability of cache.

Table 2. Key platforms characteristics

Characteristic	Altera	NVIDIA	MALI
Board	Nallatech 385	GTX 980	ODROID XU-3
Chip	5SGXA7	GM204	T628
Technology (nm)	28	28	28
Frequency	Variable	1216 MHz	600MHz
Compute Units	Variable	16	4
Floating Point Units	256	2048	16
Local Memory	6.25MB	256 KB	Virtual
Cache	-	2MB	256KB
Work-Items	Single preferred	$1024 \times 1024 \times 64$	$256 \times 256 \times 256$

3 Platform-Independent, Application-Specific Optimizations

Here we define a platform-independent micro-architecture optimization flow and apply along with application-specific optimizations applied to various use cases.

3.1 Platform-Independent Optimizations

Irrespective of underlying hardware, we target the following optimizations based on general principles of parallel computing and the OpenCL model (Fig. 2):

Explicit Parallelism: The first step is to define parallelism explicitly using the OpenCL model of work-items and work-groups. This requires a general understanding of the application to explicitly divide each task into fine-grained multiple parallel units, forming the base kernel.

Cores: The second step is to implement core-level optimizations in two ways: firstly, by providing enough parallel work-items and data to enable maximum cores utilization in time and secondly, exploiting vector operations in a single core, if available, using OpenCL vector data types for maximizing spatial usage.

Memory: The third step optimizes memory access for reduced latency by hiding the main memory access latency. This is achieved by memory coalescing and also by maximising the use of high speed, local memories in OpenCL.

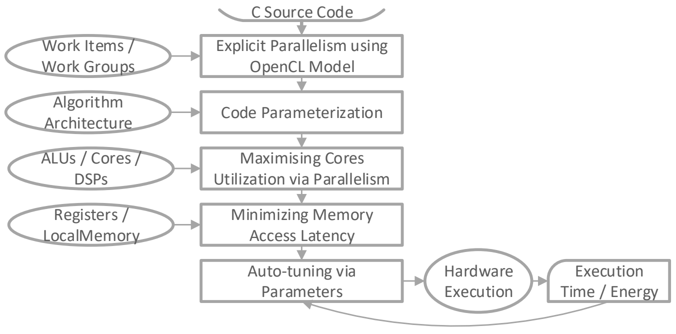

Fig. 2. Formalized Micro-architecture optimization flow

Auto-tuning: Finally, auto-tuning using an iterative approach, where the tunable parameters are scaled from minimum to maximum in power of two, is essential to maximise workload balancing and resource utilization. In our context, this involves varying OpenCL parameters such as number of work-items, work-groups, loop unrolling, etc., via argument passing to compiler, to find the optimum combination.

Next, we consider these optimizations in addition to application-specific optimizations that are applicable to all platforms.

3.2 Application-Specific Optimizations

We start by applying the steps of optimization flow on SGEMM and then briefly summarize the applied optimizations on BOP.

Matrix-Matrix Multiply: 4 different implementations of SGEMM kernels have been investigated. Let us consider the pseudo code for the kernel as:

```
for (i in range n)
  for (j in range m )
    acc = 0
    for (k in range p)
      acc += A(i,k) * B(k,j);
    C(i,j) = acc;
```

Then each kernel optimizes the pseudo code as follows:

Kernel 1, the basic implementation, distributes the first two outer loops into parallel work-items such that each work-item computes one element of C. There is plenty of parallelism but no explicit use of data locality.

Kernel 2 exploits on-chip, fast memory by loading smaller blocks of data on local memory and maximally utilizing it before being replaced by new data from global memory.

Kernel 3 builds on kernel 2 to exploit the fastest memory, registers. Using the same local memory of the OpenCL model, the inner-most loop is divided into sub-blocks, targeting data-locality equal to the size of the registers.

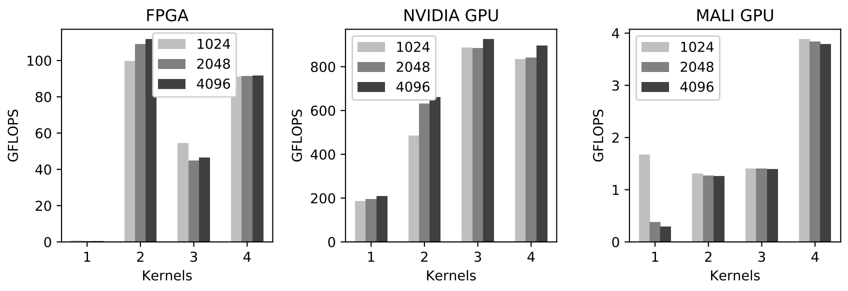

Fig. 3. SGEMM throughput variations on devices for square matrices of various sizes mentioned in legend

Kernel 4 exploits high bandwidth memory operations as well as parallel arithmetic operations, if supported by hardware units. This is achieved by processing vector of values in SIMD fashion using vector data types of OpenCL.

Furthermore, block size for kernels 2-4 is chosen to make maximum use of local memory and data-locality. All devices benefit from memory coalescing when consecutive work-items access data from consecutive memory locations. Finally, auto-tuning for SGEMM is simple, as each work-item works in a single instruction multiple data (SIMD) fashion on elements and data size is large enough to keep resource usage to maximum.

Binomial Option Pricing: The BOP computation is not as inherently parallel as SGEMM as the iterations over number of steps during backwards walk have loop carried dependences. Within each step, there is anti-dependence due to each value being used in two backward nodes calculation. The later can be removed by using two buffers. Parallelism can also be achieved by pricing multiple options in a work-group.

Targeting the defined flow, the first basic implementation included definition of kernel's parallelism via work-groups and work-items. From the top level, the number of options in a work-group can be varied with each work-group running on an individual CU. To maximise local memory usage, maximum number of options are selected such that all intermediate values are stored on local memory.

Similarly, multiple work-items per option operate on different nodes at a single time step in parallel and thus balance cores usage. However, increasing number of options per work-group or number of work-items per option put constraints on local memory bandwidth as work-items share local memory. Work-items also need to synchronize before moving on to next step in backward walk due to true dependence and the penalty for that increases with increasing number of work-items. In addition, fewer work-items limits the parallelism available to maximise compute resource usage. The optimum point is achieved with auto-tuning as explained in Section 4.1. Vector processing is also evaluated.

Finite Difference Time Domain: FDTD kernels from vendor optimized libraries were only auto tuned for each device to maximise local memory and cores usage.

4 Throughput Analysis

In this section, we analyse the achieved throughput computed via kernel processing time on the 3 targeted accelerators using the defined optimization flow.

4.1 Throughput Variations

SGEMM: We observe varying trends for all devices for SGEMM kernels 1-4 (Fig. 3). For kernel 1, the FPGA performs the worst compared to GPUs. This can be partially attributed to more mature GPU compilers allowing them to scale out significant performance from basic definition of explicit parallelism. More importantly, although no local memory usage is defined for any device, the cache in GPUs is able to improve memory latency whilst FPGAs suffer due to non-availability of cache.

Throughputs for kernel 2 improve for all platforms. However, FPGA benefits the most of this optimization due to large local memory of FPGA and its explicit description compensating for lack of cache. For kernel 3, both the GPUs perform better than the 2^{nd} kernel, making use of the registers. For FPGA, the performance degrades as explained later. For kernel 4, only the Mali GPU is able to exploit vectorization. For Altera and NVIDIA, the additional control instructions provide overhead and perform worse than kernel 2 and 3, respectively.

SGEMM - FPGA Analysis: We take a more detailed look at various SGEMM kernels mapping on hardware to better understand the variation of the throughput on FPGA. Looking at Table 3, kernel 1 offers reasonable parallelism, defined by loop unrolling, and a decent operating frequency. However, the actual throughput is lower than expected due to global memory latency.

Kernel 2 achieves maximum parallelism and full utilization of DSPs. Using only two nested loops results in lower overhead and the highest synthesized frequency. The theoretical performance after synthesis is about 120 GFLOPs compared to actual 111 GFLOPs which is due to higher global memory latency.

Kernel 3 is interesting since, unlike GPUs, it degrades throughput. This is due to lower frequency owing to the additional overhead of 3^{rd} nested loops over the sub-blocks. In FPGAs, the compiler should use registers for 3^{rd} loop elements, however, the OpenCL memory architecture does not support a different memory type for registers and the Altera compiler is not able to do it automatically. Also the structure of kernel 3 made full utilization of DSPs difficult, resulting in 192 way parallelism only.

Finally, an extra processing dimension in kernel 4 due to two way vectorization of elements in matrices A and B resulted in the lowest frequency. This requires a loop to go over the width of vector with each iteration multiplying a vector element from A with a sub-vector element (selected via switch statement) of vector B. However, it offers improvement in memory latency owing to vectorized access and achieves the same performance as predicted after synthesis.

Surprisingly, Altera, NVIDIA and Mali perform the best for Kernels 2,3 and 4 and show up to 298×, 4.4× and 13× improvement over their worst implementation, respectively.

Table 3. SGEMM synthesis results on FPGA

Resource	Kernel 1	Kernel 2	Kernel 3	Kernel 4
Logical Elements (%)	28.87	41.41	36.11	45.59
Flip Flops (%)	22.61	30.43	26.31	38.43
RAMs (%)	62.38	49.80	33.55	60.66
DSPs (%)	39.06	100	84.37	100
Frequency (MHz)	210.7	236.23	201.93	179.14
Parallelism	32	256	192	256
GFLOPS	0.37	111.65	46.43	91.66

BOP: As described earlier, after the parametrized implementation of the BOP kernel, auto-tuning is needed to optimize number of work-items in a work-group. For all devices, the throughput followed a similar trend and increased when the work-items were increased by power of two starting at 4. Taking 2048 steps as an example, the best throughput for NVIDIA, FPGA and Mali GPUs was seen at 128, 128 and 64 work-items, respectively. After that it started increasing again for higher number of work-items due to resources contention.

Vectorization performed worse for Altera and NVIDIA while for Mali the 4-way vectorisation improved the throughput by more than 3×. Using more number of options per work-group reduced performance owing to increased local memory bandwidth contentions. Even if all other optimizations are ignored, Altera, NVIDIA and Mali showed 2.8×, 3.5× and 2× improvement over worst case via balancing of work-items.

4.2 Throughput Comparison

The absolute throughputs for varying data sizes are shown in Fig. 4. We chose the best throughputs for each device which were measured using OpenCL's clGetEventProfilingInfo. As expected, the NVIDIA GPU performs the best by up to 8×, 17× and 56× over Altera FPGA for BOP, SGEMM and FDTD respectively. The FPGA performs up to 56×, 5.5× and 16× better than Mali GPU. The overall performance on all devices can be related to the size of each device.

4.3 Theoretical vs Achieved Throughput

Compiler efficiency can be estimated by analysing their ability to achieve high throughput compared to the theoretical peak performance on each platform. The theoretical peak throughput for NVIDIA GPU is 4612 GFLOPS computed as number of $cores \times Frequency \times 2(ArithmeticPipelinesPerCore)\ FLOPS$. As the Stratix V has 256 of 27×27 and 512 of 18×18 multipliers and a floating point unit uses 2 of 27×27 or 4 of 18×18 multipliers with 2 FLOPs per cycle with a peak operating frequency of 300 MHz [9], the peak throughput can be calculated as $256 \times 300 \times 2 = 153.6\ GFLOPS$. For Mali GPU, the peak is

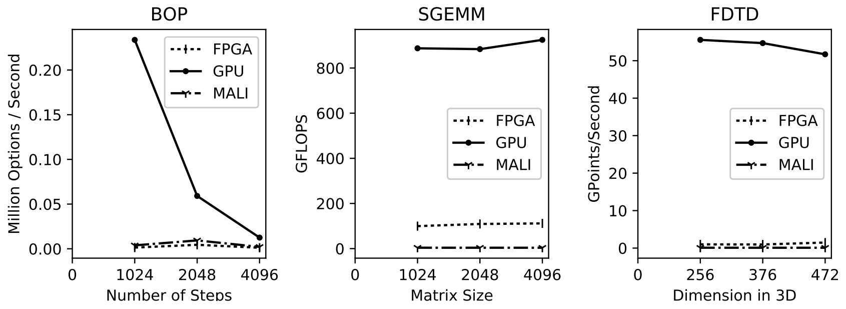

Fig. 4. Throughput comparison across devices.

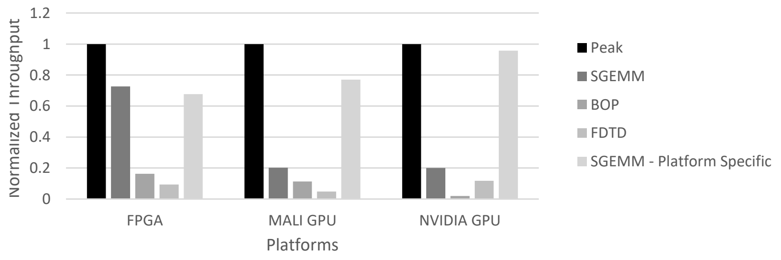

Fig. 5. Normalized Peak vs achieved throughput for various devices

estimated using $cores \times Frequency \times 2(ArithmeticPipelinesPerCore) \times 4(way-vectorization)$ $FLOPS$ as 19.2 $GFLOPS$.

For SGEMM the FLOPs are calculated as $2 \times n^3$ where n is the size of square matrices. For BOP, the total FLOPs are given by $3 \times n(n+1)/2$, where n is number of steps. For FDTD, we used a radius of 4 for calculating the new field values. A radius of 4 constitutes 25 points and thus 49 FLOPs per point. The total number of FLOPs are then $dimx \times dimy \times dimz \times 49$.

We have also included figures for achieved performance using platform-specific optimizations for SGEMM on each platform. For FPGA and MALI GPU, figures are projected estimates from implementations in [10] and [11], respectively, on similar devices while NVIDIA figures are of execution via CuBLAS library [12]. The normalized peak throughputs are shown in Fig. 5. It shows that for our used methodology, FPGA performs the best and even though GPU is supposed to be the preferred candidate for OpenCL, it requires more optimization effort utilizing platform-specific characteristics to achieve maximum throughput.

5 Energy Efficiency Analysis

To focus only on energy consumed in computing, we measured the dynamic power, i.e. the power utilized on top of static power during computation, using on-board sensors and looked at how it varied for different use cases.

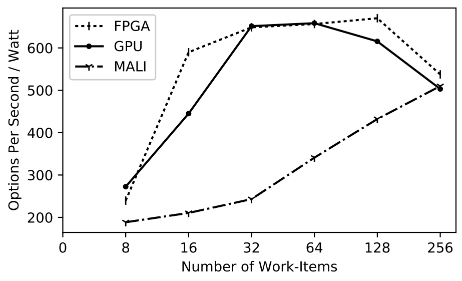

Fig. 6. BOP energy efficiency variation with number of work-items

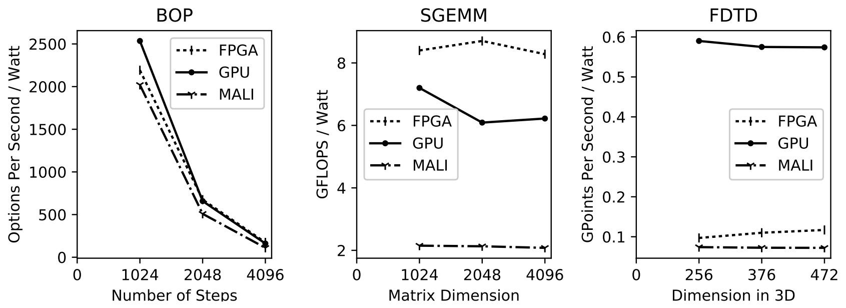

Fig. 7. Energy efficiency comparison across devices

5.1 Energy Efficiency Variations

SGEMM: Although the energy efficiencies of all devices for SGEMM followed a similar pattern to the throughput as both depend on resource usage, they are not exactly proportional. For example, kernel 4 outperforms kernel 3 for NVIDIA GPU, kernel 3 is better than kernel 4 for FPGA while kernel 2 is better than kernel 3 for Mali GPU whereas it was vice versa for throughput. In addition, although FPGA has the best peak performance, NVIDIA is better than FPGA for kernels 1 and 3 highlighting the importance of the platform-independent optimizations. Overall, the optimization methodology improves energy efficiency of FPGA, NVIDIA and Mali by 24× (Kernel 2), 5.8× (Kernel 3) and 14× (Kernel 4), respectively, compared to the worst performing points.

BOP: The achieved energy efficiency trend for 2048 steps (Fig. 6) is different from throughput with a maxima of 64 and 256 work-items for NVIDIA and Mali GPU, respectively. Interestingly, the energy efficiency curves cross over at multiple points for all platforms. Overall, the methodology improves energy efficiency of Altera, NVIDIA and Mali by up to 2.8×, 2.7× and 2.7× , respectively.

5.2 Energy Efficiency Comparison

As with throughput analysis, we choose the best energy efficiency points for each device which may not have the best throughputs. The graphs shown in Fig. 7 against application-specific metrics are self explanatory to characterize the accelerators based on energy efficiency. For BOP, Altera performs 1.15× worse than NVIDIA for a smaller problem size but performs up to 1.02× better for higher step sizes. For SGEMM and FDTD, Altera and NVIDIA perform the best by up to 1.4× and 6×, respectively. Mali performs the worst for all cases.

6 Conclusion

This work develops a design space exploration based on OpenCL and the identification of platform independent optimization that are applied on a variety of accelerators. To evaluate the energy efficiency and performance of the considered accelerators we mapped three popular application kernels. The results show that although GPUs outperform FPGA in terms of throughput, FPGA is able to achieve better energy efficiency for some of the tested cases whilst not requiring traditional hardware design expertise. On the other hand GPU requires more platform-specific optimizations utilizing in-depth knowledge of hardware to achieve high performance.

Acknowledgment

The work was supported by the European Commission under European Horizon 2020 Programme, grant number 6876281 (VINEYARD).

References

1. E.E. Schadt, et al. "Computational solutions to large-scale data management and analysis", in *Nature Reviews Genetics*, 2010, 11(9), pp.647-657.
2. J.E. Stone, D. Gohara and G. Shi, "OpenCL - A parallel programming standard for heterogeneous computing systems", in *Computing in science and engineering*, 2010, 12(3), pp.66-73.
3. J. Barr, "Developer PreviewEC2 Instances (F1) with Programmable Hardware." *Amazon Web Services*, 2016.
4. K. Hill, et al. "Comparative analysis of OpenCL vs. HDL with image-processing kernels on stratix-v fpga," in *IEEE Int. Conf. on ASAP*, 2015.
5. J. Fang, A. L. Varbanescu, and H. Sips,"A comprehensive performance comparison of CUDA and OpenCL," in *IEEE ICPP*, 2011.
6. S. Rul, et al. "An experimental study on performance portability of OpenCL kernels," in *Symp. on Application Accelerators in High Performance Computing*, 2010.
7. D. Chen and D. Singh, "Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms," in *ASP-DAC*, IEEE, 2013.
8. H. R. Zohouri, et al. "Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs," in *Proc. IEEE/ACM Supercomputing Conf.*, 2016
9. I. Berkeley Design Technology, "Floating-point DSP design flow and performance on altera 28-nm fpgas," in *Independent Analysis*, 2012.
10. H. Giefers, R. Polig, and C. Hagleitner, "Analyzing the energy-efficiency of dense linear algebra kernels by power-profiling a hybrid CPU/FPGA system, " in *25th Int. Conf. on Application-specific Systems, Architectures and Processors*, IEEE, 2014.
11. J. Gronqvist, and A. Lokhmotov, "Optimising OpenCL kernels for the ARM Mali-T600 GPUs," in *GPU Pro 5: Advanced Rendering Techniques*, pp-327, 2014.
12. NVIDIA, CUDA, "Basic Linear Algebra Subroutines (cuBLAS) library," 2013.